Technical Report
CMU/SEI-87-TR-10
ESD-TR-87-111

(2)

A Guide to the Classification and Assessment of Software Engineering Tools

Robert Firth
Vicky Mosley
Richard Pethia
Lauren Roberts
William Wood

August 1987

Technical Report
CMU/SEI-87-TR-10
ESD-TR-87-111
August 1987

A Guide to the Classification and Assessment of Software Engineering Tools

Robert Firth
Vicky Mosley
Richard Pethia
Lauren Roberts
William Wood

Tools and Methodologies for Real-Time Systems Engineering Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

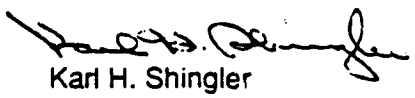
SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER



Karl H. Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Table of Contents

A Guide to the Classification and Assessment of Software Engineering Tools

Abstract

Software engineering tools are computer programs that assist people in doing the work of software engineering. As understanding of the software engineering process has broadened and the need to solve problems has intensified, there has been increasing interest in using software engineering tools. Understanding what a tool does and comparing it to similar tools are difficult tasks given the diversity of functionality that exists. This report describes a tool classification technique that helps those investigating tools decide where a tool fits in the software engineering process and identify what a tool does or doesn't do. It also provides guidance to the tool evaluation process and lists specific criteria that should be considered when evaluating tools.

1. Introduction

The Tools and Methodologies for Real-Time Systems Engineering Project at the Software Engineering Institute (SEI) is exploring and evaluating tools and methodologies that support the process of software development at various stages in the software development cycle. The intended audience for the project's work is project leaders, program managers, and procurement managers who are directly concerned with implementing requirements for real-time embedded systems, selecting the most suitable software engineering environments to implement these systems, or assessing contractor proposals on the basis of the feasibility and risk of the engineering principles the contractors propose. This document describes one of the project's ongoing activities: the classification and assessment of software engineering tools. It presents a tool classification and description mechanism that is being used by the project to gain a better understanding of the large number of publicly available and supported tools as well as the relationship between methods and tools. It also outlines a set of evaluative criteria that apply across a range of tools. The classification mechanism, evaluative criteria and assessment process outlined in this report are being used by SEI personnel and industrial affiliates participating in the project to support the ongoing activities of the project as well as the evaluation of individual tools.

Tools play an important role in the software engineering process, and the proper selection and use of tools can aid those involved in the development of real-time systems. Increased understanding and use of tools will occur in individual organizations in an incremental fashion over time. Tool selection will be done by those involved in the process of software engineering to meet varying needs while, at the same time, living with organizational and project constraints. Tool selection in that case can be a complicated and confusing process.

While this document focuses primarily on tools, it is important to remember that tools are based on engineering methods and techniques and provide automated assistance to engineers using those techniques. Tool classification and assessment should only be conducted by an organization after it has gained an understanding of software engineering methods and has determined which methods it will adopt. Issues involved with understanding, evaluating and using methods will be discussed in separate SEI publications where they can be dealt with in depth. While the classification and assessment of methods and tools are related, the issues are sufficiently different that they should be treated separately.

This report outlines the project members' view of software engineering tools and provides a framework for organizations thinking about and evaluating tools. The report pertains to a broad range of tools and does not restrict itself to tools that are relevant to only the process of engineering real-time systems.

The following four chapters of the report primarily address these questions in turn:

- What are software engineering tools and what value do they have?
- How can tools be thought about to minimize confusion and simplify the selection process?
- What general tool attributes are important to consider in the selection process?
- What is the general process to follow in evaluating and selecting tools?

2. Software Engineering Tools

Software engineering is the systematic approach to the specification, development, operation, maintenance, and retirement of software [2]. The process of real-time software engineering is expensive, time consuming, and labor intensive. Those involved in the process are faced with a myriad of problems. Symptoms of the problems are widely recognized and, from a product and project standpoint, include missed schedules, cost overruns, and systems with low reliability, poor performance, and less functionality than expected.

Those who have been dealing with the general problems of engineering production, efficiency, and quality have recognized that improvement in these areas can come from analysis and refinement of the complex system of engineering process, management, and methodology. They also recognize the value of tools in dealing with the problems. Tools are broadly defined as devices for doing or facilitating work. Tools in and of themselves have little value. A tool is valuable in a specific situation because of two basic characteristics: the tool does something relevant to the situation; the tool assists human labor by automating aspects of useful methods and techniques.

Software engineering tools are devices that assist people in doing the work of software engineering. For the purpose of this report, a tool is defined as: "a computer program used to help develop, test, analyze, or maintain another computer program or its documentation; for example, automated design tool, compiler, test tool, maintenance tool" [2]. The notion of using tools in the production of software is not new. *Devices such as editors, assemblers, compilers,* linkers, debuggers, etc., are tools that are widely used and recognized as valuable. As the understanding of the software engineering process has broadened and the need to solve problems has intensified, there has been increasing interest in developing and using software engineering tools as well as capital investment in them.

The increased interest in tools is demonstrated by two major and very visible activities:

1. The development of a commercial marketplace for software engineering tools. This marketplace is currently very active with new vendors, new products, and significant product upgrades occurring on a regular basis. Academic, professional, and commercial conferences are devoting increased time and space to presentations, demonstrations, and evaluations of software engineering tools.
2. Increased academic research and commercial activities in the area of definition, production, and evaluation of software engineering environments: sets of integrated hardware and software devices that support the process of software engineering.

While the evolution of environments promises increased productivity and improvements in product quality, there is much work to be done in this area and many questions are yet to be answered. Many organizations have a pragmatic need to improve their capabilities now and to do so in a way that does not force them into the position of wholesale replacement of their existing facilities, methods, and tools. Many existing organizations have invested considerable amounts of time, money, and effort in developing and evolving their software engineering environments. Engineers have been trained in the use of these environments and in the methods that these

environments support. These organizations can neither justify nor afford to quickly and completely replace these environments. As stated earlier, many organizations will improve their understanding and use of tools in an incremental fashion over time. The commercial tools marketplace demonstrates that tool and tool-set product offerings will evolve toward integrated software engineering environments.

3. Software Engineering Tools Taxonomy

A taxonomy is a classification scheme. Taxonomies have been used in a number of areas and are useful as the subject matter in an area becomes increasingly diverse and complex. The balance of this section discusses a taxonomy of software engineering tools.

Project members believe that a taxonomy of software engineering tools is necessary to support the work of the Tools and Methodologies for Real-Time Systems Engineering Project. They also believe the taxonomy will assist organizations that are actively involved in evaluating and selecting tools for use in their particular projects. As SEI project members surveyed the tools marketplace and gathered information on available tools, they were faced with a large number of tools that were described in a variety of different ways. Understanding what a tool does and comparing it to similar tools is a difficult task given the diversity of functionality and description that exists today. The need to deal with this diversity led project members to develop this taxonomy and its associated description mechanism. They have provided a simple, consistent means to classify and describe a tool. The purpose of this mechanism is to allow them and others to quickly classify a large number of tools. The classifications can then be used to identify a subset of those tools for further evaluation.

The specific goals being addressed include the following:

1. As individual tools are investigated, the taxonomy helps to decide where the tool fits in the software engineering process, which activities it supports, and how it supports them.
2. The taxonomy provides a framework for the discussion of individual tools.
3. The taxonomy helps to characterize and record what a tool does or doesn't do.
4. The taxonomy helps to clarify which parts of the software engineering process have been addressed by those creating tools and which parts of the software engineering process are deficient in tool support.
5. The taxonomy helps to clarify one's thinking about tools.

Project members created the taxonomy because no taxonomy had been found that addresses these needs sufficiently, provided useful information, and was not cumbersome to use and understand. Existing articles and publications have dealt with the issues of classifying tools [1] [3] [5] [6]. These publications have stated explicitly that they are classification schemes, or they have attempted informally to classify tools. The problems that project members have had in directly applying existing classification schemes to their work include the following:

1. The taxonomies have been overly detailed and therefore confusing. More work is involved in understanding and interpreting the taxonomy than in understanding a tool to be classified.
2. The taxonomies use terms that are not well defined or commonly understood.
3. The taxonomies have mixed classification with evaluation and selection criteria, i.e., they have mixed the issues of what a tool does with how the tool does it and how well it does it.

To be useful, a taxonomy should have a certain set of characteristics. These characteristics include the following:

1. The taxonomy should be simple and straightforward. It should encourage looking at tools at a high level and discourage thinking about large amounts of detail. The taxonomy should not be a comprehensive listing of tool features, but instead should give a broad idea of whether or not a tool should be considered by a particular firm with a particular application.
2. A large number of people should be able to understand and use the taxonomy. It should be based on a small number of well-defined concepts that are familiar or can easily become familiar to a wide audience.
3. It should separate issues of engineering methodology from tool classification. Many tools support particular methodologies, and selecting a methodology suited to the engineering task at hand is certainly a key decision. Methodology and tool selection is, however, a two step process: select the methodology first, then select supporting tools. Attempting to deal with both methodology and tool issues at once confuses the selection of both.
4. It should concentrate on what a tool does and where it fits into the software engineering process.
5. It should easily accommodate existing and future tools.

Project members created the taxonomy with these considerations in mind. A two-dimensional matrix format was chosen. One dimension is the phase in the software development cycle during which the tool is typically used; the other is the kind of operation the tool performs upon objects. A project management column is included since project management activities need to be addressed across the entire development cycle. Space is included in the taxonomy for objects and development cycle phases that were not mentioned, but that a prospective tool user might need to incorporate.

Although the taxonomy provides a means to easily think about an individual tool's place and role in the development cycle, it does not by itself provide enough information to someone who is interested in purchasing the tool. Therefore, project members have added tool descriptions and other information that might play an important role in tool selection, such as the methodology (if any) supported, required hardware, and price range. The taxonomy is meant to serve as a guide and aid for someone wishing to select tools for an application. Therefore, the taxonomy is flexible, and the user should feel free to add information, or delete information that is not pertinent.

By looking at the completed taxonomy, one may find where in the software development cycle the tool will be used and what things the tool could do. A potential tool user can complete the tool taxonomy and the accompanied information in a short period and then answer important questions that aid in deciding whether the tool may be suitable for an application. Candidate tools that appear to be appropriate can then be chosen and looked at in more depth.

The next two pages outline a taxonomy and tool description format, including brief descriptions of required information. To aid the users of the taxonomy, all terms used in the taxonomy have been defined in sections 3.1 and 3.2. In addition, examples of completed classifications appear

in Appendix A to this document. When classifying a tool, it will be useful to refer to these definitions and examples, as the terms used in the taxonomy may mean different things to different people.

Tools Taxonomy

Tool Name - Name of tool classified.

Version Identification/Release Date - Version number and release date of the tool.

Tool Supplier - Vendor supplying the tool, including address and phone number.

Brief Description - One- or two-sentence description of tool.

Development Phase / ***Operation on Object***	***Project Management***	***System/SW Req'ts Analysis***	***SW Requirements Analysis***	***Preliminary Design***	***Detailed Design***	***Coding and Unit Testing***	***CSC Integration and Testing***	***CSCI Testing***	***Sys Integ/Test***	***Other***
create										
transform										
group										
analyze										
refine										
import										
export										
other										

Special Definitions

Place X's in the boxes of the taxonomy to indicate tool functionality in a phase of the development cycle.

Definitions for the row and column labeled "other" should be included here if they are used.

Objects and Operations

For each object the tool deals with, describe the object and each of the operations performed on the object.

Attributes

- ***Required Computer/OS*** - Hardware and operating system (if needed) for tool operation
- ***Other Required Hardware*** - Any other required hardware such as graphics monitors, mouse, or bit pads
- ***Other Required Software*** - Any other required software such as a spreadsheet program or database manager
- ***Methodology Supported*** - Methodology(s), if any, supported by the tool
- ***Other Tools Interfaces*** - Other tools that can be integrated or used in conjunction with the tool
- ***Price Range*** - Price ranges are: Low (up to $10K); Medium ($10 to $30K); High (above $30K). Ranges were chosen as a means to represent the cost of one copy of the tool as prices vary relative to such variables as the buyer, supplier, and quantity purchased.

Comments

This section should include other information not reflected by the taxonomy and the objects and operations descriptions.

- Special features that are especially noteworthy
- Comments or notes on user interfaces
- Constraints on the tool, for example, works only for the Ada programming language
- Tool set structure (what components comprise the tool)
- Basis for the tool classification: vendor supplied, tool user, review of brochures, or product documentation
- Tool history
- Other.

3.1. Software Development Cycle

The software development cycle is the period of time that begins with the decision to develop a software product and ends when the product is delivered [2]. It is defined as an iterative process, divided into phases, with a certain set of activities conducted and work products produced at each phase [4].

Different sources name each phase in a slightly different fashion; however, for the purpose of defining the tools taxonomy and focusing this work, eight of the phases defined in DOD-STD-2167 were chosen. These phases include: system/software requirements analysis; software requirements analysis; preliminary design; detailed design; coding and unit testing; computer software component (CSC) integration and testing; computer software configuration item (CSCI) testing; and system integration and testing. These divisions are not disjoint; there is an overlapping that occurs across the phases as well as an iterative feedback loop between the phases.

Tools used during these phases generally support technical software engineering activities. Tools supporting the activities of project management must also be included. Therefore, the development phases are prefaced with an additional category for Project Management. Because each subculture of the software industry has its own representation of the development cycle and some tools may not "match" the given phases, an additional category labeled "Other" has been provided on the taxonomy form.

The decision to base one of the axes of the taxonomy on DOD-STD-2167 phases was made for two reasons. First, the phases provide a model of the software development process that covers a broad range of activities. Second, the model is familiar to a large number of people. Our use of the model focuses on the activities that typically occur during each phase and on the objects that are typically created in each phase rather than on the sequence of phases prescribed by the model. Each phase is defined by its objectives, associated activities, and created objects. DOD-STD-2167, the *IEEE Standard Glossary of Software Engineering Terminology*, and the SEI's *Methodology for the Evaluation of Ada Environments* document [7] served as a basis and guide for these definitions.

3.1.1. Project Management

Although not a phase of the software development cycle, project management is a set of activities that spans all phases. The software development cycle represents a process, and, like all processes, it must be managed to be effective. For the purpose of this document, configuration management activities have been included within project management. The activities involved with project management are the following:

- project estimation (staffing and cost)
- project planning (scheduling, critical path analysis, resource allocation)
- status reporting and progress tracking
- resource usage and cost tracking
- information collection and data management

- communications (within and between projects)
- quality assurance (reviews and audits)
- configuration management (configuration control and version control)
- risk analysis
- standards verification
- productivity measurement
- presentations
- document referencing
- office support.

Project management tools are known to create and manipulate such objects as Pert and Gantt charts and other forms of documentation such as spreadsheets, tables, graphs, and text.

3.1.2. System/Software Requirements Analysis Phase

During this phase, system requirements are refined through studies, system engineering activities, analysis, and development of prototype hardware and software. Requirements are allocated to equipment and software, and interfaces are defined. The major activities of this phase include:

- system requirements definition, analysis
- partitioning of requirements to hardware and software
- prototyping of hardware and software
- system timing/sizing analysis.

Objects created during this phase include system/segment specifications, prime item development specifications, preliminary interface requirements specifications, and preliminary software requirements specifications for each major software component.

3.1.3. Software Requirements Analysis Phase

During the software requirements analysis phase, system needs are studied to arrive at a refined definition of software requirements. These requirements are generally derived from system requirements. Software requirements (functional, performance, and interface) are defined at a top level, and detailed requirements are developed through active coordination, analysis, review, and planning. Analysis is accomplished independently of any implementation language. The major activities of this phase include:

- requirements definition, analysis, decomposition
- tradeoff studies
- rapid prototyping (for algorithm analysis, man-machine interfaces analysis)
- system performance simulation
- timing/sizing analysis

- test software requirements definition
- software quality standards and measures definition
- software development planning.

Examples of tools that could be used during this phase of the life cycle include prototyping tools, system-level simulators, data flow diagramming and analyzing tools, state transition diagramming and analyzing tools, and timing and sizing analyzers. The objects that are created, analyzed, and refined by these tools are generally textual and graphical.

3.1.4. Preliminary Design Phase

The allocation of the requirements to top-level functional components is the primary objective of this phase. The processes that take place in these functional components are described as well as the interface relationships, including data flows and control flows, between the components. The deliverables in this phase include specification documents that prescribe — in a complete, precise, and verifiable manner — the requirements, architecture, design, behavior, or other characteristics of a system or system component. This phase generally terminates with the successful completion of a preliminary design review (PDR). Other, more general activities include:

- software test plan description
- software requirements and interface requirements specifications
- software requirements tracing/tracking
- top-level software design document creation and analysis

Examples of tools that could be used during this phase include requirements specification languages (a formal language with special constructs and verification protocols used to specify, verify, and document requirements) and tools supporting structured analysis (SA) methods. Objects of these tools are primarily documentation and graphic diagrams.

3.1.5. Detailed Design Phase

The objective of the detailed design phase is to refine the design approach by developing a detailed design for the software solution — including a more detailed definition of the interfaces among units and a detailed procedural flow. The functions to be performed to meet the requirements of the specifications are detailed. Detailed definitions of components, modules, interfaces, test approach, and data for the software system are created and reviewed. It is an iterative process usually resulting in changes to the specification and corresponding changes in the design. This phase generally terminates with the successful completion of a critical design review (CDR). Other activities of this phase include:

- design analysis and decomposition
- design walkthroughs
- prototyping
- completion of detailed software design document, interface design document, and database design document

- test procedures completion.

Examples of tools and/or objects that can be used in this phase of the life cycle include program design languages (PDL), prototyping tools, structured design (SD) tools, data flow diagrams (DFD), control flow diagrams (CFD), data dictionaries (DD), and other design analyzers (tools that accept information about a program's design and produce such objects as module hierarchy diagrams, graphical representations of control and data structure, and lists of accessed data blocks).

3.1.6. Coding and Unit Testing Phase

The objective of this phase is to implement the design generated in the previous phase. The software product (code) is created from the design documentation, unit tested, and debugged. Activities of this phase include:

- code generation and translation
- code walkthroughs or peer group reviews.
- debugging
- unit testing (at the module level)

Tools that support this phase of the life cycle include language support software (e.g., compilers, linkers, assemblers), symbolic debuggers, context editors, instruction set simulators, and module test case generators. The objects of these tools are generally code (source, object, and executable code).

3.1.7. Computer Software Component Integration & Testing Phase

The objective of this phase is to check the execution of aggregates of software against the requirements agreed upon in the development specifications and to determine whether all modules function together per the specified requirements. The emphasis is on the interaction between software units through their interfaces in order to demonstrate complete processing functions and validate performance requirements for each software program component. This may include reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, services, or documents conform to specified requirements. Quality assurance is a planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements.

Testing is usually performed in increments — as more and more of the software units are completed, they are integrated with additional modules and tested. Integration testing is often performed by an independent organization that is both technically and managerially separate from the organization responsible for developing the product or by a group other than that which executed the original design. Activities of this phase include:

- preparing and evaluating integration test procedures
- conducting formal quality tests
- producing test reports.

Tools that support this phase include symbolic debuggers, emulators, test analyzers, test report generators, and test scenario generators. The objects these tools operate on are executable code modules.

3.1.8. Computer Software Configuration Item (CSCI) Testing Phase

The CSCI testing phase is performed to ensure the proper operation of a major software component. Objectives and activities are the same as the previous phase except that the activities are performed on major components rather than smaller aggregates of software modules.

3.1.9. System Integration and Testing Phase

The system integration and testing phase is the period in the software development cycle during which major software components are integrated with each other and all software is integrated into its operational environment. Testing of the entire system is performed to ensure it performs in a manner that satisfies its specified requirements. Activities include:

- requirements verification and validation
- review of test plan, procedures, and results for compliance to standards
- system integration testing
- system integration problem identification and test reporting
- documentation verification.

Examples of tools that could be used during this phase include environment simulators, scenario generators, test data analyzers, and requirements trackers/tracers.

3.2. Objects and Operations on Objects

An object is an entity that has a particular purpose or function and a set of defined operations that create and manipulate it. The word "objects" was chosen to classify the different kinds of products that can be generated by a tool. The form of the object and the operations that can be performed upon the object help to distinguish it from other kinds of objects.

Objects and operations on objects play two roles in the classification and description mechanism described in this report. First, Operations on Objects are used as one axis of the tool's taxonomy. This provides a vehicle for those classifying tools to quickly capture high-level information regarding what a tool does or does not do. It also allows those reviewing completed classifications to determine if a tool may be at all relevant to their needs and decide whether or not the tool should be investigated further.

The second, and major, role played by the concept of objects and operations on objects is as a vehicle to concisely describe what a tool does. Tool descriptions and tool documentation often describe the functionality of a tool in a variety of different ways. Occasionally, documentation focuses on the output of a tool and describes few of the operations required to achieve the output. Conversely, documentation often describes tool functionality in a way that does not clarify

what specific objects are created or manipulated by the tool. Objects and operations on objects can be used to identify both basic aspects of a tool's functionality: what operations it performs and what persistent objects it performs them on.

The following are descriptions and examples of objects and operations performed upon them.

3.2.1. Create Objects

This operation allocates appropriate system resources to represent and contain the object, while providing a means to access the object. It may also create some or all of an object's structure and content. Within a tool, a text or data object may be created, for which the tool will allocate space and store that object. In other cases, the tool will allocate space and provide a template or skeleton that is defined for that kind of object. Some tools will allow you to provide definitions for templates or complete objects and may be able to create entire objects automatically from other objects. Examples of objects are:

- data flow and other graphical diagrams
- documentation texts (such as specifications)
- macro or template definitions
- rule definitions upon which analysis can be done
- code
- data structure definitions
- test data and test scenarios
- analysis output.

3.2.2. Transform Objects

This operation produces a new object, one with changed content or format, from an existing object. Transformation between objects in a tool set is important because it implies that the tool user can transfer information across tools. Examples of transformation on objects include:

- generating documentation or reports from such things as diagrams, data structures, and analysis output
- generating other diagrams or generating a skeleton or template for a diagram from a diagram that has already been defined
- generating code from a set of design objects (e.g., generating code from diagrams and/or a design language).

3.2.3. Group Objects

This operation establishes relationships between objects or classifies objects. Sometimes this will result in the group of objects being treated as a new object itself. A tool user may want to organize or categorize objects. In most instances, a new object or objects will be created as a result of this operation. Examples are:

- combining separate documentation objects and/or diagrams to form a complete documentation object

- grouping definitions of objects such as data structures or test-case definitions
- grouping code objects, such as linking them together and compiling
- grouping objects from a separate software system for use in a new software system.

3.2.4. Analyze Objects

This operation studies or dissects an object by measuring it against some analysis criteria. Analysis can be done on a particular object itself or on a number of objects. The result of this operation is usually an output object (e.g., analysis report) or a new, "corrected" version of the original object. Examples of analysis operations are:

- checking within an object or among objects for consistency
- checking for syntax, punctuation, or spelling
- checking an object for completeness
- checking failure possibilities
- checking for unreachable paths
- checking for test coverage completeness.

3.2.5. Refine Objects

This operation changes or adds to an object often as the result of performing analysis upon the object. Examples are:

- adding additional detail to a design specification
- adding stubs or defaults for undefined data structure components
- constructing code templates for exception handling.

3.2.6. Import/Export Objects

This operation imports an object from an external environment into the current one and, conversely, exports an object from the current environment to a different one. It may also provide specific functions to move objects between individual tools (e.g., import/export from design tools to configuration management database). Examples of import/export operations are:

- downloading or uploading objects from one piece of hardware to another
- transferring an object from one grouping or environment to another
- acquiring a component from a library of reusable components.

4. Tool Evaluation

A taxonomy is not an evaluation. The former assigns a tool a place in a classification matrix to give an indication of what the tool does and where it is used. The latter attempts to assess how well the tool does its job relative to the needs of the evaluator.

Such evaluations are inevitably somewhat subjective since everyone has different requirements, works in a different environment, and has different ideas about how tools ought to work. However, many *questions* that a potential user asks about a tool can be standardized, while accepting that different users will interpret the answers in different ways and attach different degrees of importance to them.

The following sections discuss these questions, grouped according to the aspect of a tools acquisition, support, and performance they address. These aspects are:

1. Ease of Use
2. Power
3. Robustness
4. Functionality
5. Ease of Insertion
6. Quality of Commercial Support

The first four sections are mainly of concern to the actual user of the tool; the last two are of concern to the management of the project that contemplates acquiring the tool.

The questions listed apply across a wide variety of tools and cover areas that must be considered when evaluating any tool. Each question is phrased such that a positive response indicates a positive tool attribute. A discussion of using these questions as the basis for a tool assessment process is provided in Chapter 5 of this document.

4.1. Ease of Use

One measure of a tool's effectiveness is the ease with which the user can interact with it. Clearly, no matter how functional or complete a tool is, if the user spends most time thinking about how to use the tool or making the tool work, then the tool is hindering and not helping with the task. To justify using a tool, the tool's benefits must offset its cost and the time spent using it.

4.1.1. Tailoring

Tailoring is an important aspect of a tool. A tool can be used by a wide variety of organizations and users. If a tool can be tailored to user needs or to a particular user style, the tool has the potential to be used with more dexterity and at a faster rate than would be otherwise. While tailoring can provide positive benefits, it is important to recognize that indiscriminate tailoring can disrupt team efforts when each user tailors the tool to an individual style.

1. Can various aspects of the interface be tailored to suit user needs, including application and ability level?

2. Can the user define new commands or macros for commonly used command sequences or chain macros together?
3. Can the user "turn off" unwanted functions that might be obtrusive?
4. Can the tool's input and output formats be redefined by the user?
5. Can tailoring operations be controlled to maintain consistency within the using project/organization?
6. Can the tool be configured by the user for different resource tradeoffs to optimize such things as response speed, disk storage space, and memory utilization?
7. Does the vendor support and assist tailoring the tool to the specific users needs?

4.1.2. Intelligence/Helpfulness

A tool helps the user by performing particular functions. The more intelligent a tool, the more functions it will perform without the user having to directly specify their initiation. In addition, the tool should anticipate user interaction and provide simple and efficient means for executing functions the user requires.

1. Is the tool interactive, for example, does it prompt for command parameters, complete command strings, or check for command errors?
2. Is action initiation and control left with the user?
3. Is quick, meaningful feedback on system status and progress of interaction and execution given to the user?
4. Is the interface simplified by the use of sound or graphics (icons, color coding, shape, texture, etc.)?
5. Can the user access and retrieve stored information quickly and with little effort while using the system?

4.1.3. Predictability

Unpredicted responses from the tool usually result in unhappy users and unwanted output. Command names should suggest function, and users should rarely be surprised by a tool's response. If an unpredicted response does occur, the user should have a means to "undo" the command. If the result of a particular command has drastic results, the user should be warned before the command is actually executed.

1. Are the responses from the tool expected in most cases?
2. Is it possible to predict the response of the tool to different types of error conditions?

4.1.4. Error Handling

Not only should the tool be tolerant of user errors, it should check for and correct these errors whenever possible.

1. Does the tool recover from errors easily?
2. Does the tool protect the user from costly errors?
3. Does the tool periodically save intermediate objects to ensure that all work is not in vain if a failure occurs during a long session of tool use?

4. Does the tool protect against damage to its database caused by inadvertent execution of the tool?
5. Does the tool help the user correct errors?
6. Will the tool check for application-specific errors, such as checking if parentheses match?

4.1.5. System Interface

Not only is it useful for a tool to interact with one user, but it may be appropriate for a tool to accommodate interaction with many users or other tools.

1. Is the tool designed to be used by more than one person at a time?
2. Does the tool provide for management, including access control, of work products for single and multiple users?
3. Is the interface compatible with other tools in a tool set or other commercially available tools?
4. Does the tool provide for output devices such as printers?

4.2. Power

What does one mean by the *power* of a tool? Here are some examples concerning a tool that nearly everyone uses, a text editor. A powerful editor can, for instance:

- globally replace "HAL" with "DEC"
- recognize "love" and "love?" as two instances of the same word, but correctly recognize "glove" as another word
- warn that "neccessary" looks wrong
- automatically indent paragraphs, inserting new lines between words at the appropriate points
- automatically number paragraphs or sections, renumbering after insertion or deletion.

The power of a tool seems to arise from two main sources:

- the extent to which the tool "understands" the objects it is manipulating
- the extent to which simple commands can cause major effects.

In addition, a tool can give the impression of greater power by keeping more knowledge about its internal state such as a command history. Power is also demonstrated by reasonable performance achieved through efficient use of the computing resource.

4.2.1. Tool Understanding

The objects that a tool manipulates have an inner structure. Structured objects tend to be comprehended in terms of two things: the framework and the particular content. For example, I := J is read as an assignment statement (framework) that assigns J to I (content). This text is being read as the introduction (framework) to a section that explains tool understanding (content).

Hence, questions can be asked about the extent to which a tool understands the structure and its content, and also about the ability of the tool to handle more detailed or more general aspects of that structure.

1. Does the tool operate on objects at different levels of abstraction or at different levels of detail?
2. Can one zoom in or zoom out from one level to another?
3. Can the tool modify collections of objects so as to preserve relationships between them?
4. Can the tool remove objects and repair the structure, and insert objects with proper changes to the new structure?
5. Does the tool perform any validation of objects or structures?
6. Can the tool automatically create structural templates?
7. Can the tool expand objects into templates of the next level of detail, with consistency checking between levels?

4.2.2. Tool Leverage

Leverage is the extent to which small actions by the user create large effects. The leverage of any interactive tool is a function of its command set. The usual way to increase this leverage is to allow a user to define *macros* — short commands that stand for longer command sequences. Another way, more in keeping with an object-oriented view of the world, is to define a command as an action to be applied to a specific object. Commands can then be "overloaded," i.e., the same command name can have a different implementation for different objects. Commands can also be inherited, composed, and so on.

To illustrate this, consider a command **print** applied to a fragment of a parse tree. One print style can be defined for expressions and another for comments (overloading). If an attempt is then made to print a commented expression, the right thing (composition) should be obtained automatically.

If this facility is missing, a tool can be made to do more only by multiplying the number of commands, e.g., having a **printcomment** command and a **printtree** command. Most people would agree that this does not make a tool more powerful; the increase in the number of things it can do is matched by a corresponding increase in the effort the user has to expend to learn, remember, and select the commands. Indeed, since such extensions to the command set typically address more and more marginal areas of the requirement, "creeping featurism" dilutes the power of the tool.

1. Can commands be bound to specific object types or structure templates?
2. Can commands be applied systematically to entire collections of similar objects?
3. Can polymorphic commands be applied to entire structures that contain diverse objects?
4. Can commands be executed indefinitely until a predicate is satisfied?

4.2.3. Tool State

This is an inductive approach to increasing the power of a tool. If a tool remembers how it has been used in a current session or in previous sessions, it can provide the user with simpler ways of invoking effects by saying, "Do to this what you just did to that."

1. Does the tool keep a command history?
2. Can commands be reinvoked from the history?
3. Can the command history be saved to be replayed by a new run of the tool?
4. Can the reinvoked commands be modified when they are replayed?
5. Can one save the current state of the tool and the objects it is manipulating, and subsequently restore it?
6. Does the tool learn, i.e., does it keep state across invocations?
7. Does the tool keep and/or employ statistics of command frequency and operand frequency?

4.2.4. Performance

The performance of a tool can greatly affect the ease with which it is used and can ultimately determine the success of a tool within an organization. A tool must be able to function efficiently and be responsive to the user. Poor tool performance can create costs that negate many of the benefits realized from tool use; a tool that performs inefficiently may result in missed schedules or frustrated users who are skeptical that the tool really helps them.

1. Is the tool's response to commands acceptable relative to the complexity of the operations performed by the command? For example, is the user waiting for unreasonable amounts of time, or is there any response lag on simple or frequently used commands?
2. If the tool supports multiple users, is response and command execution time acceptable with the maximum load of users?
3. Can the tool, running on the user's hardware, handle a development task of the size required by the user?
4. Does the tool provide a mechanism to dispose of any useless byproducts it generates?

4.3. Robustness

This section is concerned with the robustness of the tools operating on a system. The robustness of a tool is a combination of such factors as: the reliability of the tool, the performance of the tool under failure conditions, the criticality of the consequences of tool failures, the consistency of the tool operations, and the way in which a tool is integrated into the environment.

While the robustness of individual tools is important, it is secondary to the robustness of the environment in which the tools operate. Although the tool and the tool set of which it is part can be robust and consistent, many characteristics of robust operation are best done on a more global environment where the tool writer has to worry about correct interfaces to the environment.

but does not have to be concerned with a great many services that are provided by the environment to maintain system integrity. For example, most tools should not be concerned with security issues, access authorization, archiving, device interfaces, etc. These should be handled by the environment in which they are embedded. The tool should be concerned with having the correct interfaces to be inserted in the environment and to operate properly within the environment. The issues described in the following sections are tool-related robustness issues.

4.3.1. Consistency

These issues are concerned with the consistency of operation of the tool.

1. Does the tool have well-defined syntax and semantics?
2. Can the output of the tool be archived and selectively retrieved and accessed?
3. Can the tool operate in a system with a unique identification for each object?
4. Can the tool re-derive a deleted unique object, or does the re-derivation create a new unique object?
5. Does the tool have a strategy for dealing with re-derivation of existing objects, such that it finds the objects rather than re-deriving them? (This has important consequences on the performance characteristics of the system.)

4.3.2. Evolution

In all but the most unusual cases, tools evolve over time to accommodate changing requirements, changes to the environment, correcting detected flaws, and performance enhancements. The questions below are related to the evolution of the tool.

1. Is the tool built in such a way that it can evolve and retain compatibility between versions?
2. Can the tool smoothly accommodate changes to the environment in which it operates?
3. Can new versions of the tool interface with old versions of other related tools?
4. Can new versions of the tool operate correctly on old versions of target objects?
5. Can old versions of the tool operate correctly on new versions of the target objects?
6. Can separate versions of the tool coexist operationally on the system?
7. Has the tool been implemented on/ported to various hosts?
8. Can the tool's output be interchanged between hosts?

4.3.3. Fault Tolerance

There are many ways of defining fault tolerance. This work is not concerned with the general problem, but with fault tolerance that specifically is related to individual tools.

1. Does the tool have a well-defined atomicity of action?
   (**Note:** This does not necessarily mean that each invocation of the tool must have an atomic effect on the system. It simply means that no intermediate states should be registered, and that any environmental failures during execution of the tool do not cause irreparable damage once the failure has been repaired and the system restarted.)

2. If the tool is found to be incorrect, can the system be rolled back to remove the effects of its incorrect actions?

4.3.4. Self-Instrumented

A tool is a piece of software performing a function and, like all other software, may have various types of bugs or flaws associated with it at any point in its life cycle. Most bugs are detected during testing and deployment, but there are often latent bugs remaining after deployment, and maintenance activities are well known to introduce bugs. For these reasons, a tool must be self-instrumented to assist in determining the cause of a problem once the symptom has been detected.

1. Does the tool contain any instrumentation to allow for ease of debugging?
2. Are there tools for analyzing the results collected by the instrumentation?
3. Does the tool contain self-test mechanisms to ensure that it is working properly?
4. Does the tool record, maintain, or employ failure records?

4.4. Functionality

The functionality of a tool is not only driven by the task that the tool is designed to perform but also by the methods used to accomplish that task. Many tools support methodologies. The accuracy and efficiency with which the tool does this can directly affect the understandability and performance of the tool, as well as determine the quality and usefulness of tool outputs. In addition, a tool that generates incorrect outputs can lead to frustrated users and extra time expenditures needed to "fix" tool outputs. These additional costs may weigh heavily against tool benefits.

4.4.1. Methodology Support

A methodology is a systematic approach to solving a problem. It prescribes a set of steps and work products as well as rules to guide the production and analysis of the work products. Automated support for a methodology can aid its use and effectiveness. However, it must be made clear that the following questions *do not deal with assessing a particular methodology.* Methodology assessment should occur separately from and prior to the assessment of tools that support the methodology. The questions presented here deal with how well a tool automates and supports a methodology, not with the methodology itself.

1. Does the tool support one or more methodologies that meet the users' needs?
2. Does the tool provide a means to integrate other methodologies?
3. Does the tool support all aspects of the methodology?
4. If some aspects are excluded, are the important parts or concepts of the methodology (parts that are either important to the methodology itself or important to the development project) supported?
5. Does the tool support the communication mechanisms of the methodology (such as a textual or graphical language) without alteration?
6. Does the tool build in any functionality, in addition to direct support of the methodology, that is useful?

7. Is the tool free of functionality that is useless or a hinderance?
8. Does the tool flexibly support the methodology, for example can the user initially skip or exclude some parts of the methodology and return to it later?
9. Does the tool provide an adequate scheme to store, organize, and manipulate the products of the application of the methodology?
10. Does the tool provide guidance to ensure that the concepts of the methodology are followed when using the tool?

4.4.2. Correctness

To be useful, a tool must operate correctly and produce correct outputs. A tool evaluation must pay special attention to this critical area.

1. Does the tool generate output that is consistent with what is dictated by the methodology?
2. Does the tool check to see if the methodology is being executed correctly?
3. Is there no case where data items entered by the user are unintentionally or unexpectedly altered by the tool?
4. Are executable outputs generated by the tool "bug free"?
5. Are outputs generated by the tool correct by all standards?
6. Do transformations executed by the tool always generate correct results?

4.5. Ease of Insertion

An important aspect of tool use that is often overlooked is the ease with which a tool can be incorporated into the target organization or environment. Management and users need to be aware of how well the tool fits within the existing environment and accept changes that the tool may inflict upon the environment in which they work. Questions on ease of insertion fall into the categories listed below.

4.5.1. Learnability

Depending upon how complex it is, learning how to use a tool can result in considerable expense, time, and frustration. Not only should the tool's command set be consistent and understandable; the tool should interact with the user to help learn to use the tool properly.

1. Is the complexity of the tool proportional to the complexity application; i.e., does the tool simplify a problem rather than complicate it?
2. Do prospective tool users have the background necessary to successfully use the tool?
3. Can the users use the tool without memorizing an inordinate number of commands?
4. Do the interactive elements imply function in the problem domain, i.e., do command names suggest function or graphical symbols representative of function?
5. Are the commands and command sequences consistent throughout the system?

6. Can the user quickly do something to see what happens and evaluate results without a long set-up process?
7. Can the results, i.e., the work products produced, of learning exercises be disposed of easily? For example, can they be removed from a database without action by a database administrator?
8. Is the tool based on a small number of easy to understand/learn concepts that are clearly explained?
9. Does the tool provide a small number of functions (commands, directives) that allow the user to do the work the tool is intended to do?
10. Can the user learn a small number of simple commands initially, and gradually add more advanced commands as proficiency is developed?
11. Does the tool provide the user with templates or other aids to guide interaction?
12. Is there a method of using a help facility that aids the novice user by providing a step-by-step description of what to do?
13. Is the time required to understand and become proficient in using the tool acceptable:
    - for the average user?
    - for the average project manager?
    - for the project team?

4.5.2. Software Engineering Environment

Successful use of a tool requires a fit between the tool and the environment in which it will be used.

1. Is the tool in some ways similar to what the organization currently does and knows, for example, is there some commonality in the underlying method, process, vocabulary, notation, etc?
2. Is the command set free of conflict with the command set of other tools the organizations uses, i.e., same or similar commands with different actions?
3. Does the tool run on the hardware/operating system (OS) the organization currently uses?
4. Is installing the tool a simple, straightforward process?
5. Does the tool use file structures/databases similar to those currently in use?
6. Can data be interchanged between the tool and other tools currently employed by the organization?
7. Can the tool be cost-effectively supported by those responsible for maintaining the environment?

4.6. Quality of Support

Without adequate commercial support, a tool may become useless before it is used. The quality of commercial support connotes many things: it ranges from the cost of maintenance agreements to the level of training required and provided. When evaluating a tool, one should also consider its "track record." The evaluator should be aware of the past performance and uses of the tool as well as the past support the vendor has or has not provided.

4.6.1. Tool History

What is the tool's track record?

1. Does the tool have a history that indicates it is sound and mature?
2. Has the tool been applied in a relevant application domain?
3. Is a *complete* list of all users that have purchased the tool available?
4. Is it possible to obtain evaluations of the tools from a group of users?

4.6.2. Vendor History

Often one can infer the quality of the tool and the quality of support for the tool by looking into the track record and reputation of the vendor selling the tool.

1. Is there a staff dedicated to user support?
2. From talking to others who have had experience with the vendor, does the vendor live up to commitments, promises?
3. Are the projections for the future of the company positive, for example, does the company's future appear stable?

4.6.3. Purchase, Licensing, or Rental Agreement

1. Is the contract or agreement explicit enough so that the customer knows what is or is not being acquired?
2. Is there a cost reduction for the purchase of multiple copies?
3. Is there a corporate site license available?
4. If the user wishes, can the tool be leased?
5. Does the user have the ability to return the tool for full refund during some well-defined, reasonable period of time?
6. Is the customer given full rights and access to source code (in the event the vendor goes out of business, no longer supports the tool, and is unable to sell off rights to the product)?
7. Is the user free of all obligations to the vendor regarding use or sale of the objects generated by the tool?

4.6.4. Maintenance Agreement

1. Does a warranty (written guarantee of the integrity of the product and of the vendor's responsibility for the repair or replacement of defective parts) exist for the tool?
2. Can the user purchase a maintenance agreement?
3. Can the vendor be held liable for the malfunctioning of the tool?
4. Will maintenance agreements be honored to the customer's satisfaction in the case that the vendors sell out?
5. Is the frequency of releases and/or updates to the tool reasonable (e.g., fast enough to respond to problems, slow enough not to overburden the user with change)?
6. Does the maintenance agreement include copies of releases/updates?
7. Is the turn-around time for problem or bug reports acceptable?

4.6.5. User's Group/User Feedback

1. Does a user's group (or similar group that addresses problems, enhancements, etc., with the tool) exist?
2. Does the vendor provide a responsive, helpful hot-line service?

4.6.6. Installation

1. Is the tool delivered promptly as a complete package (*object code, documentation,* installation procedure, etc.)?
2. Does the vendor provide installation support/consultation?

4.6.7. Training

1. Is training available?
2. Has prerequisite knowledge for learning and use of the tool been defined?
3. Is training customized for the acquiring organization and individuals with attention paid to the needs of different types of users (engineers, project managers, etc.)?
4. Do the training materials or vehicles allow the user to work independently as time permits?
5. Is the user provided with examples and exercises?
6. Are vendor representatives (marketing, sales, service, training) product knowledgeable and trained?

4.6.8. Documentation

1. Is the tool supported with ample documentation (e.g., installation manuals, user's manuals, maintenance manuals, interface manuals, etc.)?
2. Is on-line help available?
3. Is a tutorial provided?
4. Does the documentation provide a description of what the tool does ("big picture view") before throwing the user into the details of how to use it?

5. Is the documentation
    - readable
    - understandable
    - complete
    - accurate
    - affordable?
6. Does the documentation have an indexing scheme to aid the user in finding answers to specific questions?
7. Is the documentation promptly and conveniently updated to reflect changes in the implementation of the tool?

5. Assessment Process

The process of assessing or evaluating a tool must be carried out by the organization that wishes to acquire the tool. As stated earlier, a tool in and of itself has no value. It is valuable only when applied by a particular individual or organization. Individuals and organizations are different; what is appropriate to one organization or individual may be inappropriate to another organization or individual.

The assessment process involves four steps:

1. Perform a needs analysis.
2. Perform an analysis of the existing environment.
3. Develop a list of candidate tools and acquire descriptions of those tools.
4. Apply assessment criteria and select a tool for use.

5.1. Needs Analysis

The first step in assessing a tool is to decide the purpose for which the tool will be used. Tools are valuable because they do something: perform a function, save time, save labor, save money, make something possible that is otherwise difficult or not possible. What the tool does must be relevant to the acquiring organization and must bring utility to that organization. In particular, the tool must do something that is appropriate to the larger task at hand: generate a document; analyze a specification; manage an object collection. It must contribute to a process controlled by a method. The organization should consider the following:

1. What is the relevant model of software development?
2. What major tasks does that process require, both technical and managerial?
3. Which tasks could be performed or assisted by automated tools?
4. Which of those tasks currently lack adequate tool support?
5. What is the estimated benefit to be obtained from specific new tools?

The organization should be clear about its software development process, methods and management, and the needs they imply, before deciding to acquire tools.

5.2. Environment Analysis

The second step of the assessment process, which can usually be done in conjunction with the needs analysis, is an analysis of the environment in which the tool will be used. Tools do not operate in a vacuum. A tool will be successfully used only if it fits in the environment of a particular organization. Each organization deals with its own environment, and those making decisions in the organization must deal with the environmental constraints.

Constraints come in different forms but can generally be classified into several different areas: economics, time, personnel skill level, vendor relations, current practice, etc. It is important to

understand the environment and the constraints the environment places on decisions made in the organization. It is also important to remember that there are two ways to deal with constraints: live with them or change them. The environmental analysis should not only identify constraints, it should also identify those constraints that can realistically be removed or changed as well as the tradeoffs involved in relaxing the constraints.

Questions to consider when performing this analysis include those on the organization's readiness to support the introduction and use of a new tool.

1. Is the organization open to change? Have changes occurred in the past? Have there been past successes or failures?
2. Have there been lessons learned from past successes or failures? Do the lessons support introduction of the tool?
3. Can the organization afford to buy the tool easily, or will the purchase place extreme pressure on learners for instant success?
4. Is the investment in the tool so large that it will be difficult to dislodge in the future?
5. Is there a plan to introduce the tool? Does everyone understand the plan? Are goals, objectives, benefits, risks, and milestones clear to all? Are all people willing to support the plan?
6. Is there an agreed-upon way to determine progress in use of the tool?
7. Is management planning to reinforce progress and initially hold back negative judgment?
8. Is the tool sponsored by a champion — someone able and willing to serve as sponsor and focal point, and to monitor and encourage progress?
9. Is training scheduled to allow real use of the tool shortly after completion of training?
10. Will those who need to learn the tool be able to do so in a low-pressure environment?
11. Will learners have adequate access to the tool during the learning period?
12. Will learners have pilot projects on which to practice use of the tool?
13. Will learners have time to experiment?
14. Has a case been made for increasing benefit over time as users become acquainted with the tool and increasingly exploit its power?

5.3. Develop Candidate List

Once an organization identifies its needs, it should develop a list of candidate tools that may meet those needs. As recognition of the value of software engineering tools has increased, a number of vendors have entered the expanding market with product offerings. Trade shows, trade publications, and technical journals provide information on available products.

The candidate list should be developed as close to the selection date as possible. The tool market is currently very active with new vendors, new products, and significant product upgrades

occurring on a regular basis. Information on products becomes stale quickly, and timely information is key when deciding on the most appropriate, available tool.

The candidate list should contain tools that, on the basis of a first brief look, seem probably appropriate to the perceived need and the organizational environment. It should exclude tools that clearly do not meet the need or that cannot function without unacceptable changes in the existing environment. The classification scheme (taxonomy and description) outlined in Chapter 3 of this document provides a mechanism to structure this first brief look. It allows those who use it to quickly capture and organize information on existing tools and to identify those tools that are probably appropriate. A blank classification form has been included as Appendix B of this document.

The candidate list should look beyond individual tools. It may be that, while no one tool does all that is required, a set of related, compatible tools can be found that jointly cover the requirement. The organization must decide how important it is to have all or most tools developed by the same vendor or with the same characteristics; in our view, there are great advantages in acquiring a tool set with a consistent philosophy - the burden of acquisition, training, support, and use is substantially less.

5.4. Apply Criteria and Select

The final step in the assessment process requires applying a set of assessment criteria to each of the candidate tools and analyzing the results. The approach suggested is motivated by earlier SEI work [7] and consists of the following four phases:

1. **Establish evaluative criteria.** The criteria listed in previous sections detail attributes that should generally apply across a broad range of tools. The criteria we give are simple, numerous, and unweighted. Each organization should look through the criteria and make its own estimate of their relative importance. For example, some organizations cannot afford the time and cost of extensive training, so they will prefer tools that are easier to learn. Others might prefer more powerful tools, even if they are harder to learn and use.

   These criteria must be augmented with the results of the organization's needs analysis and environmental analysis before a final selection is made. The resulting list of questions, ranked in order of importance, serves as the basis for the next phase.

2. **Define specific experiment.** The questions established in the first phase must be translated into tests that will be performed on each of the candidate tools. Each of the questions will result in one or more specific tests tailored for the individual tools as required. Each test should detail what is to be performed under what set of initial conditions. It should also detail what data are to be collected and what quantities should be measured to answer the underlying questions.

3. **Execute the experiment.** The tests should be conducted through hands-on use of the tool. While many of the questions could be answered through review of product literature and documentation, literature can occasionally be misleading or be misinterpreted. The tests should be sequenced according to the ranked list of criteria created in the first phase. Early tests with unacceptable results indicate that the tool will not meet the organization's critical needs. These early results can be

used to shorten the testing process and allow the organization to place increasing focus on the most promising tools.

The result of this phase should be a transcript of the execution of the experiment and the measurements and answers to the questions that were detailed in the previous phase.

4. **Analyze the results.** The final phase requires the analysis of the data collected from the experiments. The analysis should determine how well the tool satisfies each of the criteria with special attention paid to criteria ranked highest by the organization.

   Once criteria have been applied, a decision must be made. The results of the application of the criteria will usually indicate that no tool is a perfect match for all the needs and the environment of the particular organization. The final decision must be based on the judgment of those in the organization who will receive the most benefit (or harm) from the tool selection. The decision should also consider the impact of introducing the tool. Once a tool is acquired, if it is used, the software engineering environment will change. While it is not possible to predict the exact extent and nature of the change, some consideration should be given to whether or not the change will be "ecologically sound."

Finally, the assessment criteria will aid the selection process but will not, and cannot, provide a recipe for absolute success in choosing the most appropriate, useful tool. Careful, thorough assessment must be followed by planned, monitored introduction and use to increase the probability of successful adoption and use of the tool.

6. Conclusion

The taxonomy presented in this document is meant to serve as a guide for organizations interested in purchasing software engineering tools. It should provide a guide for answering pertinent questions regarding the suitability of tools for use within a particular organization, given its needs. The taxonomy itself is a tool and is subject to many of the classification and assessment criteria outlined in this document. Therefore, the taxonomy should not be cumbersome to the user and should instead help highlight tool characteristics that are important to the application.

As previously mentioned, the current market in software engineering tools reflects an increasing awareness among software developers of the benefits of employing software engineering tools. Still, a few key points about tool use should be stressed. Firstly, simply using a software engineering tool is not enough. It must be a tool that not only fits the application and needs, but fits the organization as well. Failure to understand this may result in the failure to successfully employ the tool, which may hinder an organization from trying again. In addition, the benefits from tools, like most other things, are not free. It takes a conscious effort to transition a tool into an organization, and all the activities from selection to training to tool set evolution will affect an organization's ability to effectively use the tool and reap the maximum possible benefit from it.

Appendix A: Examples

Example #1 - Tools Taxonomy

Tool Name - Computer-Aided Software Engineering Environment (CASEE)/Project Manager

Version ID/Release Date -
Ver 1.5, Jan '87 release date

Tool Supplier - Software Engineering Corporation of Tomorrow
1111 Somewhere Ave.
Anyplace, USA 00001

Brief Description - -
Project Manager is part of an integrated software tool set, CASEE. It provides a database and a set of functions to support project planning, scheduling, estimating, and management.

Development Phase / ***Operation on Object***	***Project Management***	***System/SW Req'ts Analysis***	***SW Requirements Analysis***	***Preliminary Design***	***Detailed Design***	***Coding and Unit Testing***	***CSC Integration and Testing***	***CSCI Testing***	***Sys Integ/Test***	***Other***
create	X									
transform	X									
group										
analyze	X									
refine	X									
import										
export	X									
other										

Special Definitions

N/A

Objects and Operations

Project Management

1. *Project and Task Descriptions*
    a. *creates* project and task descriptions as textual objects with defined fields for priority, precedence, and time estimate for completion
    b. *transforms* project/task descriptions and resources against a time scale automatically into a Gantt chart
    c. *transforms* tasks, priority, precedence, and time estimate for completion automatically into a Pert chart
    d. *transforms* project tasks and expended resources into status reports
    e. *refines* project and task descriptions into subprojects or subtasks
    f. *exports* descriptions in predefined report format to printer/plotter
2. *Resource Descriptions*
    a. *creates* resource descriptions such as manpower or equipment
    b. *exports* resource descriptions in predefined report format to printer/plotter
3. *Calendar*
    a. *creates* pictorial calendar with flexibility to distinguish holidays, weekends, and working days
    b. *refines calendar entries with time resolution to the hour or minute*
    c. *exports* calendar to printer/plotter
4. *Gantt chart*
    a. *creates* Gantt chart through graphic interface as well as automatically
    b. *exports* Gantt chart to printer/plotter
5. *Pert chart*
    a. *analyzes* Pert chart by determining critical path(s) of project tasks, "what if" reports when tasks and/or resources are changed, and time required to complete each task
    b. *exports* Pert chart to printer/plotter
6. *Status Reports*
    a. *analyzes* interrelationships between resources
    b. *exports* status reports via predefined report template

Attributes

- ***Required Computer/OS*** - MicroVAX/VMS, VAX/VMS[1]
- ***Other Required Hardware*** - TEK 4111[2] (1 Mbyte user buffer), mouse, printer/plotter
- ***Other Required Software*** -
- ***Methodology Supported*** - Tailorable to user's own project management methodology

- ***Other Tools Interfaces*** - Interfaces to CASEE Environment (Analyzer, Designer, Code Generator); export utility allows reports to be prepared with other word processing and spreadsheet packages
- ***Price Range*** - Low (less than $10K)

Comments

- Allows user to handle unlimited numbers of tasks and projects
- Allows graphical and textual input
- Allows personal calendar appointments/entries
- Limitation: Once tasks are defined they cannot be regrouped (they must be deleted and reentered)
- Limitation: Can only assign a maximum of six resources to a single task; resources cannot be shared across tasks
- Limitation: Does not assist with budget and cost accounting
- Basis for tool classification: Perusal of vendor brochures and user's manuals
- CASEE tool set has been on the market since July '80, has sold over 100 copies, and is in use in approximately 15 companies. The Project Manager is a newer extension of the tool set and has been released since Feb '86.

Example #2 - Tools Taxonomy

Tool Name - Computer-Aided Software Engineering Environment (CASEE)/Analyzer/RT

Version ID/ Release Date -
Ver 10.5, Jan '87 release date

Tool Supplier - Software Engineering Corporation of Tomorrow
1111 Somewhere Ave.
Anyplace, USA 00001

Brief Description -
Analyzer/RT is part of an integrated software tool set, CASEE. It contains a set of graphics editors that aid engineers in analyzing software requirements for real-time systems.

Development Phase	***Project Management***	***System/SW Req'ts Analysis***	***SW Requirements Analysis***	***Preliminary Design***	***Detailed Design***	***Coding and Unit Testing***	***CSC Integration and Testing***	***CSCI Testing***	***Sys Integ/Test***	***Other***
Operation on Object										
create			X							
transform			X							
group			X							
analyze			X							
refine			X							
import			X							
export			X							
other										

Special Definitions

N/A

Objects and Operations

SW Requirements Analysis

1. *Requirements Dictionary (RD) Descriptions*
    a. *creates* requirement descriptions in a relational requirements database
    b. *groups* requirement descriptions by keyword association
    c. *refines* requirement descriptions into subrequirements
    d. *imports* text from requirements document via optical scanner
    e. *exports* requirements descriptions to printer/plotter per predefined format
2. *Data Flow Diagrams (DFDs)*
    a. *creates* DFDs using a graphical editor to diagram top-level functional and interface requirements using processes, data stores, data flows, and externals
    b. *transforms* DFDs (processes, data stores, etc.) to DD entries
    c. *analyzes* DFDs for completeness and consistency according to Yourdon/DeMarco or Gane and Sarson methodology
    d. *refines* DFD by decomposing the diagrams to two different types of detail — process specification (final level) or a decomposed diagram (any number of levels)
    e. *exports* DFDs to a printer/plotter
3. *Data Dictionary (DD) Descriptions*
    a. *creates* DD descriptions of processes and data flows using the Backus-Naur format; *creates* process specifications using free text
    b. *analyzes* DD descriptions for consistency in the use of names and in conformance with the Backus-Naur notation
    c. *exports* DD descriptions to printer/plotter via 14 different relational formats

Attributes

- ***Required Computer/OS*** - MicroVAX/VMS, VAX/VMS
- ***Other Required Hardware*** - TEK 4111 (1 Mbyte user buffer), mouse, printer/plotter
- ***Other Required Software*** -
- ***Methodology Supported*** - Yourdon/DeMarco or Gane and Sarson for DFDs
- ***Other Tools Interfaces*** - Interfaces to CASE Environment (Project Manager, Designer, Code Generator); export utility allows reports to be prepared with other word processing packages
- ***Price Range*** - Low (less than $10K)

Comments

- Contains an integrated tool set (graphics editors, requirements dictionary, and data dictionary)
- Data Dictionary is automatically generated from DFDs — capturing all components from the graphics diagram and automatically tracking relationships between the components
- Uses mouse and menu-driven interfaces with pull-down, pop-up menus
- User can customize the symbols (icons), screen displays, text fonts, and reports
- Allows user to define rules for enforcing any user-defined methodology
- The tool set can be customized for a group of users or an individual user
- Contains an on-line help utility
- Basis for tool classification: tool user
- CASEE tool set has been on the market since July '80, has sold over 100 copies, and is in use in approximately 15 companies. The Analyzer has been on the market since September '82 with its requirements DB added in Jan '86.

Example #3 - Tools Taxonomy

Tool Name - Computer Aided Software Engineering Environment (CASEE)/Designer/RT

Version ID/ Release Date -
Ver 15.5, Jan '87 release date

Tool Supplier - Software Engineering Corporation of Tomorrow
1111 Somewhere Ave.
Anyplace, USA 00001

Brief Description -
Designer/RT is part of an integrated software tool set, CASEE. It contains a set of graphics editors which aid engineers in designing real-time systems.

Development Phase	***Project Management***	***System/SW Req'ts Analysis***	***SW Requirements Analysis***	***Preliminary Design***	***Detailed Design***	***Coding and Unit Testing***	***CSC Integration and Testing***	***CSCI Testing***	***Sys Integ/Test***	***Other***
Operation on Object										
create				X	X					
transform					X					
group										
analyze				X						
refine				X	X					
import				X	X					
export				X	X					
other				X						

Special Definitions

The "other" operation specified within the Preliminary Design Phase pertains to the tool's capability to *superimpose* Control Flow Diagrams onto Data Flow Diagrams. This operation is more than simply the creation of two transformation diagrams and the exporting of one on top of the other; the tool prompts the user for the necessary information to overlay the CFD onto the DFD.

Objects and Operations

Preliminary Design

1. *Structured Chart (SC)*

    a. *creates* SC automatically by transforming DFD processes to SC modules (as well as via a graphical SC editor)

    b. *analyzes* SC for completeness and consistency with the DFD

    c. *imports* DFD from Analyzer/RT database

    d. *exports* SC to printer/plotter

2. *Control Flow Diagrams (CFD)*

    a. *creates* CFD using a graphical editor to diagram control using control flow, control stores, and control processes

    b. *refines* CFD by decomposing the diagrams to two different types of detail — control specification (final level) or a decomposed control diagram (any number of levels)

    c. *imports* DFD from Analyzer/RT database

    d. *exports* CFD to printer/plotter or superimposed CFD/DFD onto printer/plotter

    e. *other - superimposes* CFD onto DFD using a graphical editor and responses to prompts for separating flows, processes, and stores

Detailed Design

1. *Mini Specification (MS) Descriptions*

    a. *creates* MS descriptions (free text, pseudo code, PDL, etc.) of each process at each level of the DFD using the text editor of the host environment

    b. *imports* MS names from DD

    c. *exports* MS descriptions to printer/plotter in the format pertaining to the type of description used (free text, pseudo code, etc.)

2. *Control Specifications (CS) Descriptions*

    a. *creates* CS descriptions (free text, state transition diagrams, decision tables, etc.) for each control process at each level of the CFD using a free form graphics editor

    b. *imports* CS names from DD

    c. *exports* CS descriptions to printer/plotter in the format pertaining to the type of description used (free text, diagrams, tables, etc.)

3. *Data Dictionary Data Structures*

    a. *creates* data structure descriptions for the data flows in the DD — describing data types, range constraints, alias, and textual notes

    b. *transforms* data structure descriptions into data and type declarations automatically for C and Pascal

    c. *imports* data flow descriptions from DD

    d. *exports* data structures descriptions to printer/plotter in pre-defined format

Attributes

- ***Required Computer/OS*** - MicroVAX/VMS, VAX/VMS
- ***Other Required Hardware*** - TEK 4111 (1 Mbyte user buffer), mouse, printer/plotter
- ***Other Required Software*** -
- ***Methodology Supported*** - Hatley/Lear Siegler for CFD; Constantine for SC
- ***Other Tools Interfaces*** - Interfaces to CASE Environment (Project Manager, Analyzer/RT, Code Generator); export utility allows reports to be prepared with other word processing packages; interfaces with external PDL and STD packages
- ***Price Range*** - Low (less than $10K) .

Comments

- Contains an integrated tool set (graphics editor and data dictionary)
- CFD components are automatically entered into the DD
- Uses mouse and contains menu-driven interfaces with pull-down, pop up menus
- User can customize the symbols, screen displays, text fonts, and reports
- The tool set can be customized for a group of users or an individual user
- Contains an on-line help utility
- User can alternate between the DFD and CFD to view the data and control representations together or separately
- Limitation: does not perform consistency analysis of its CFD
- Basis for tool classification: tool user
- CASE tool set has been on the market since July '80, has sold over 100 copies, and is in use in approximately 15 companies. The Designer has been on the market since September '82 with its real-time extensions added in June '86.

Appendix B: Taxonomy Format

Tools Taxonomy

Tool Name -

Version Identification/Release Date -

Tool Supplier -

Brief Description -

Development Phase / *Operation on Object*	*Project Management*	*System/SW Req'ts Analysis*	*SW Requirements Analysis*	*Preliminary Design*	*Detailed Design*	*Coding and Unit Testing*	*CSC Integration and Testing*	*CSCI Testing*	*Sys Integ/Test*	*Other*
create										
transform										
group										
analyze										
refine										
import										
export										
other										

Special Definitions

Attributes

- *Required Computer/OS -*
- *Other Required Hardware -*
- *Other Required Software -*
- *Methodology Supported -*
- *Other Tools Interfaces -*
- *Price Range -*

Comments

References

[1] Peter Freeman and Anthony I. Wasserman.
Software Development Methodologies and Ada.
Technical Report NTIS AD-A123 710/6, University of California, Irvine, November, 1981.

[2] The Institute of Electrical and Electronics Engineers, Inc.
IEEE Standard Glossary of Software Engineering Terminology.
February, 1983.

[3] Gineen Brement.
Software Life Cycle Tools Directory.
Technical Report STI 185, IIT Research Institute/Data & Analysis Center for Software, March, 1985.

[4] U.S. Department of Defense.
Defense System Software Development.
Military Standard, DOD-STD-2167.
June 4, 1985.

[5] Rome Air Development Center.
A Taxonomy of Tool Features for A Life Cycle Software Engineering Environment.
Technical Report, Command and Control Division, Griffiss Air Force Base, NY 13441, April 15, 1985.

[6] STARS Joint Service Team for Software Engineering Environment.
Preliminary System Specification, Software Technology for Adaptable Reliable Systems, Software Engineering Environment.
January 30, 1986.

[7] Nelson Weiderman.
Methodology for the Evaluation of Ada Environments.
Technical Report SEI-86-MR-2, Software Engineering Institute, Carnegie Mellon University, February, 1986.

Notes

[1]MicroVAX, VAX, and VMS are trademarks of Digital Equipment Corporation.

[2]TEK is a registered trademark of TEKTRONIX, Inc.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-87-TR-10	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-87-111
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INSTITUTE 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE 8b. OFFICE SYMBOL (If applicable) SEI JPO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY SOFTWARE ENGINEERING INSTITUTE JPO PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) See Cover	
12. PERSONAL AUTHOR(S)	
13a. TYPE OF REPORT FINAL 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) SEPTEMBER 1987 15. PAGE COUNT 64
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) SOFTWARE TOOLS, CLASSIFICATION, EVALUATION

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

SOFTWARE ENGINEERING TOOLS ARE COMPUTER PROGRAMS THAT ASSIST PEOPLE IN DOING THE WORK OF SOFTWARE ENGINEERING. AS UNDERSTANDING OF THE SOFTWARE ENGINEERING PROCESS HAS BROADENED AND THE NEED TO SOLVE PROBLEMS HAS INTENSIFIED, THERE HAS BEEN INCREASING INTEREST IN USING SOFTWARE ENGINEERING TOOLS. UNDERSTANDING WHAT A TOOL DOES AND COMPARING IT TO SIMILAR TOOLS ARE DIFFICULT TASKS GIVEN THE DIVERSITY OF FUNCTIONALITY THAT EXISTS. THIS REPORT DESCRIBES A TOOL CLASSIFICATION TECHNIQUE THAT HELPS THOSE INVESTIGATING TOOLS DECIDE WHERE A TOOL FITS IN THE SOFTWARE ENGINEERING PROCESS AND IDENTIFY WHAT A TOOL DOES OR DOESN'T DO. IT ALSO PROVIDES GUIDANCE TO THE TOOL EVALUATION PROCESS AND LISTS SPECIFIC CRITERIA THAT SHOULD BE CONSIDERED WHEN EVALUATING TOOLS.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) (412) 268-7630 22c. OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.